NISTIR 4655

USER'S MANUAL FOR THE NIST TTCN TRANSLATOR VERSION 3.0

David H. Su

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Advanced Systems Division
Computer Systems Laboratory
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4655

USER'S MANUAL FOR THE NIST TTCN TRANSLATOR VERSION 3.0

David H. Su

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Advanced Systems Division
Computer Systems Laboratory
Gaithersburg, MD 20899

May 30, 1991
August 31, 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

Table of Contents

USERS' MANUAL FOR THE NIST TTCN TRANSLATOR V3.0

David H. Su

Standard conformance testing is the first step in ensuring the interoperability of communications products. There has been increased interest in the development of Abstract Conformance Test Suites (ATS) in the standards community. A standard test script language called Tree and Tabular Combined Notation (TTCN) has increasingly been used to specify test suites. NIST has developed a TTCN to C language translator to help the industry in speeding up implementation of ATSs into executable test suites (ETS). This manual describes processes involved in implementing an ETS, how the translator is to be used, and the design of the translator itself.

Keywords: abstract test suite; data communications; conformance testing; executable test suite; protocol standard; TTCN; translator.

1. Introduction[1]

The Tree and Tabular Combined Notation (TTCN) is an international language for specification of abstract conformance test suites (ATS) for communications protocols. It is defined in Part 3 of a multi-part standard, ISO 9469 OSI Conformance Testing Methodology and Framework [1].

In order to accelerate the implementation of an ATS into an Executable Test System (ETS), the National Institute of Standards and Technology (NIST) has developed a TTCN translator that translates the TTCN test scripts into "C" programming language source statements that can easily be compiled on any target test system.

The original translator was implemented in 1989 for the DP[2] (Sydney) version of the TTCN. It has since been updated to accept the DIS (Teddington) version of the TTCN language.

This document is not intended to be a tutorial on the TTCN and C languages, as it is assumed that the reader has a basic understanding of these languages. The remainder of this document is organized as follows: section 2 presents the basic processes to implement an ETS from a TTCN ATS, section 3 shows the input/output of the NIST TTCN Translator and its output options, section 4 describes the requirements for the test system support and routines to be provided by the test system implementor in order to run the translated C code, section 5 shows the test case

[1] The identification of commercial products in the text is for clarification of specific concepts. In no case does such identification imply recommendation or endorsement by NIST, nor does it imply the product is necessarily the best for the purpose.

[2] DP (Draft Proposal) and DIS (Draft International Proposal) are terms used to denote the status of an ISO standard in its development cycle.

on the TTCN language, and lastly, section 7 presents a detailed description on the translation of TTCN test scripts into C statements.

2. Developing Executable Test System Using the NIST Translator

A TTCN ATS is represented in two forms: a human readable graphic form, called TTCN.GR, for publication of test suites; and a machine processable form, called TTCN.MP, for transferring test suites between computer systems and for processing by computer programs. After a test suite has been developed for a protocol standard, the conformance test system developer first enters the test scripts into a data base using a TTCN graphic form editor [2,3]. The machine processable form is generated from the data base using a formatter which is typically part of the TTCN editor. The test suite in TTCN.MP form is then fed into the NIST TTCN Translator. The resulting C routines are then compiled, and linked with test machine environment support routines to produce an Executable Test System. This process is depicted in Figure 1. The lines and arrows connecting boxes represent the processes and programs invoked in each step, and the boxes represent inputs and outputs of each of these processes.

The abstract test suite in TTCN form is a high level description of test sequences. For example, the TTCN Protocol Data Unit (PDU) definitions are abstract descriptions which may not necessarily match the true PDUs in terms of the size and ordering of fields within a PDU. The translated C source codes, even though executable, are just another abstract representation of the test scripts. The test system implementor must provide additional supports to make these translated C source codes executable on a real test machine. These supports are represented in the box labeled "Test System Environment Support" in Figure 1.

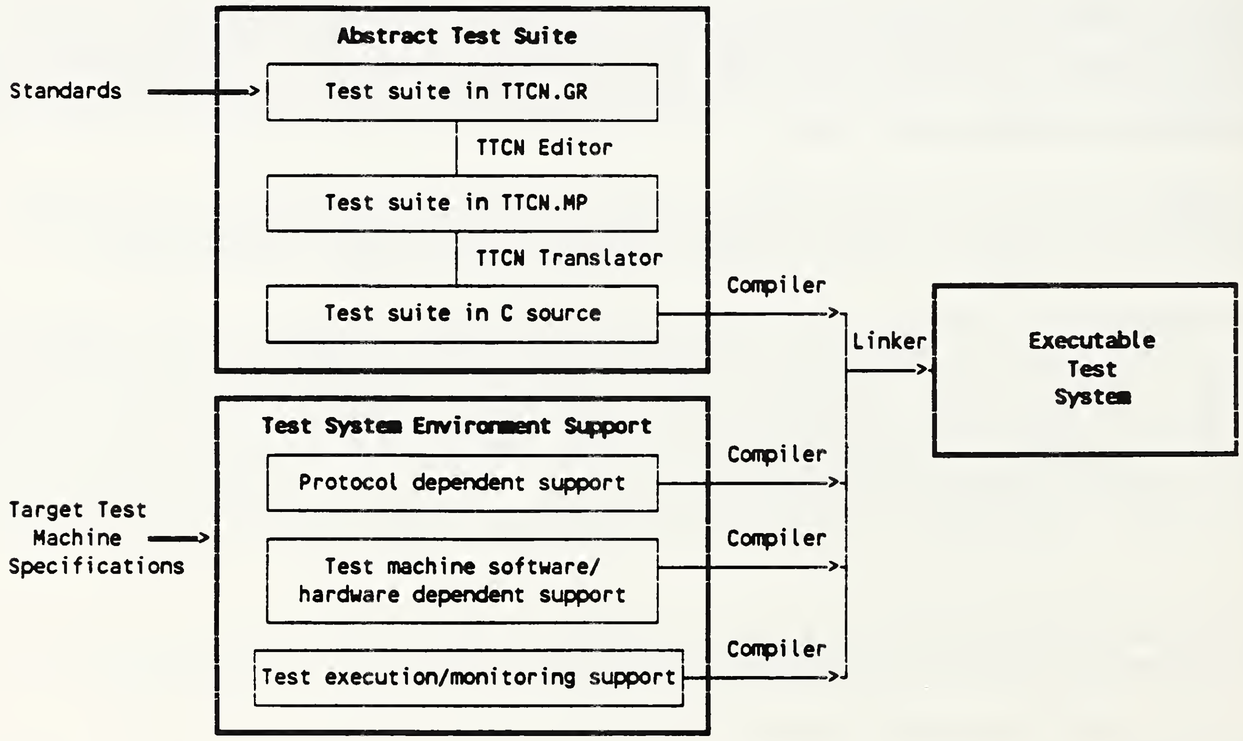

Figure 1. Conformance Test System Development

There are three types of supports needed: 1) protocol dependent supports for the mapping of abstract PDU structures to the real PDUs, 2) machine dependent supports for realization of TTCN events on the target test machine, such as sending and receiving of N-PDUs through (N-1) layers of software/hardware supports and the activation and deactivation of hardware timers, and 3) the test system user interface for the selection and execution of test cases, and for the recording of test results.

3. The NIST TTCN Translator

The translator was developed using the compiler writing tool YACC[3] and LEX programs [4,5,6] on the UNIX system, and could be run on any system that supports the C programming language. The procedures described below, however, applies to the UNIX environment only.

3.1 Input and Output of the Translator

A TTCN test suite consists of several major components: declarations of PDU structures, variables and timers, mapping of implementation specific parameters into internal variables (test suite parameters), declaration of PDU field values (PDU constraints), and test scripts (dynamic behavior tables). The TTCN translator maps these items into corresponding C language constructs and puts them into various files as instructed by the user. The header file (.h file) contains definitions for PDU structures, and external definition of all variables and routines generated by the TTCN translator, and is included by all other C program files (.c file) generated. Variables for all data structures and pointers to these data objects are declared in the .c file for test cases. The C routines for the constraints, test cases, and test steps are produced in one or more .c files depending on the options selected (see next section). The main program to run test cases, and the routine to initialize the test suite parameters, are always generated in a separate file called "driver.c" (or other name as directed). In addition, there is a set of definitions and routines in files "sysdef.h" and "sysdef.c" that are common to every test suite. Instead of generating them during every compilation, they are provided as part of the distribution tape for the TTCN translator. The following table shows the mapping of inputs and outputs.

TTCN.MP	C Source
Declarations -	Header file for all data structures
Test Suite Parameters	Variable declarations and routine "tsparameter()"
Test Suite Constants	#define statements
Variables (Test Suite)	Variable declarations
Variables (Test Case)	Variable declarations and routine "testcase_var_init()"
PDUs	Variables for PDU structures
Constraints	C routines, one per constraint/group
Dynamic Behaviors -	
Test Cases	C routines, one per test case
Test Steps	C routines, one per test step
	Main program - test case driver
	System definition and utilities - sysdef.h and sysdef.c

[3] The actual compiler generator used was Bison by Free Software Foundation [7].

3.2 *How to Run the Translator*

The translator is invoked by the following UNIX shell command:

```
                                                 -s
      ttcnc  mp-filename  [-m filename]  [-h filename]  [-g filename-prefix] [-l]
                                                 -t
```

where the square brackets [ ] indicate that the enclosed item is optional, and are not part of the command. All optional items, except -l, are specifications for output files; all filenames specified will be automatically appended with either a .c or .h suffix as appropriate. The following is the list of options:

-m : Specify the name of the file for the main routine from which the test cases are invoked. The default file is "driver.c"

-h : Specify the name of the file for the data structure header file. If this option is omitted, the filename specified in the -s option is used; if both options are omitted, the default name is "suite.h"

The next three options, -s, -g, -t, specify the grouping of the test scripts, and are mutually exclusive.

-s : All output, except the main driver and the header file, are to be stored in one file.

-g : The test cases are to be grouped based on the test suite structure, and stored in separate files. The filenames are created by concatenating the prefix specified, and the group name in the test suite. (See next selection for more details.)

In addition, the following files are created (the asterisks represent the prefix):

*decl.c - declarations for variables
*cons.c - routines for the TTCN constraints part
*lib.c - routines for the test step library

-t : Each test case is stored in a separate file. The filename is the concatenation of the prefix and the test case identifier. Just like the -g option, files *decl.c, *cons.c, and *lib.c will also be generated.

If none of the -s, -g, -t options are specified, -s is implied and all output will go to "suite.c".

-l : For each PDU field, provide space for the length of the field. (See section 3.3 for explanation.)

The input file contains the complete TTCN.MP test suite. If it is not specified, the standard input file is used.

If the test suite uses abbreviations, a preprocessor must be used to expand the abbreviated strings. In this case, the shell command is as follows:

```
        prep mp-source | ttcnc options
```

Examples:

```
    ttcnc lapd.mp
```

Compile the TTCN test suite in lapd.mp and produce files suite.h, suite.c and driver.c.

```
    ttcnc lapd.mp -m lapdtest -g lapd
```

Compile the TTCN test suite in lapd.mp and produce the following files: suite.h lapdtest.c, lapdedcl.c, lapdcons.c, lapdlib.c, and one lapd.c per group.*

```
    prep x25.mp | ttcnc
```

Expand abbreviations in test suite X25.mp and compile it to produce output files suite.h, suite.c, and driver.c.

3.3 Grouping of Test Cases to Separate Files

A typical test suite consists of hundreds of test cases, and many more routines in the test step library. If all routines generated by the translator are put into a single file, the file may become too large to manage and maintain. Furthermore, the size of random access memory (RAM) on the test equipment may not allow all test cases to be resident in RAM at the same time. The translator options -g and -t allow a system implementor to segment a test suite into separate files by group and test cases, respectively. The grouping of test cases is based on the test suite structure. An abstract test suite normally divides test cases in a tree structure into groups, and layers of subgroups, as shown in Figure 2. The -g option will structure the output in such a way that the test cases in each group will be stored in one file, and that the name of the subgroup will become part of the filename. Normally the 'C' source codes will be located in files corresponding to the lowest level of grouping. For example, assume that the lowest level of grouping of a test suite consists of S10, S20, ..., S80, and that the translator is invoked with the option "-g link", then the following files produced will contain the translated test cases: linkS10, linkS20, ..., linkS80.

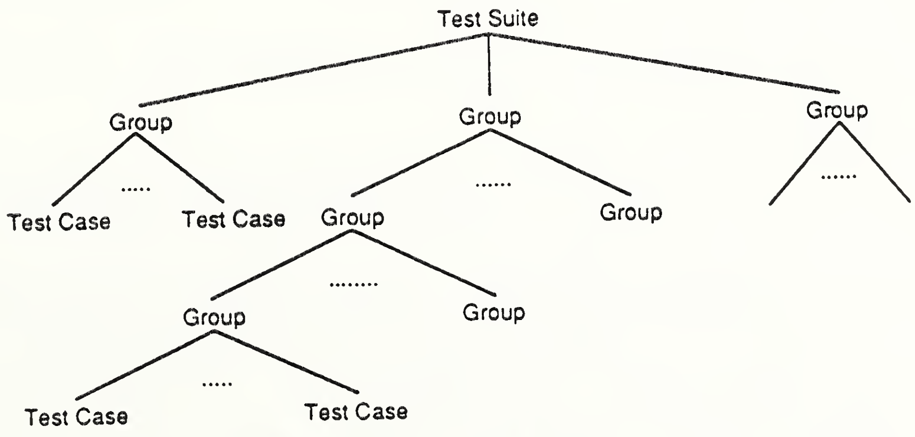

Figure 2. Tree Structure of an Abstract Test Suite

4. The Test System Environment Support

This section describes the facilities to be provided by a test system implementor on the target test machine in order to support the execution of C routines produced by the TTCN Translator.

4.1 Test Execution/Monitoring Support

A TTCN test suite provides only a collection of test cases. A test system for an ATS must also provide mechanisms to execute these test cases and to produce a test results report. The test case selection function implemented must allow the test operator to select test cases based on certain criteria: specific test case, specific groups of test cases, answers to Protocol Implementation Conformance Statement (PICS) and Protocol Implementation eXtra Information for Testing (PIXIT), etc. The TTCN translator generates a test case driver as the main program in which test cases are executed sequentially (see Section 5). This program is intended for use as a prototype, and is to be replaced by the system implementor with a new program that provides choices of test case selection criteria described above.

The monitoring function must provide a trace of PDUs received and transmitted, and records of execution results, such as test cases executed and the verdicts assigned.

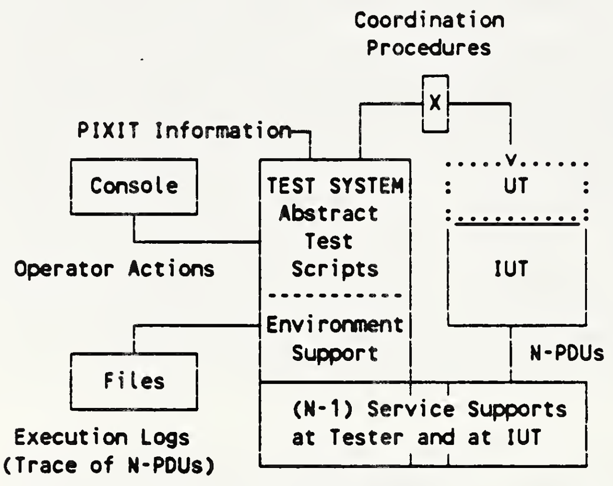
Coordination Procedures
X
PIXIT Information
Console
Operator Actions
TEST SYSTEM Abstract Test Scripts
Environment Support
UT
IUT
N-PDUs
Files
Execution Logs (Trace of N-PDUs)
(N-1) Service Supports at Tester and at IUT

Figure 3. Configuration for a Layer N Test System

It is expected that the TTCN translator will be used primarily for ATSs written for the Remote Single Layer test method, under which no specifications for the Upper Tester (UT) is provided. As a result, some kinds of test coordination procedures must be provided to successfully complete the test. Figure 3 shows the configuration of a test system for this type of ATS. The NIST Translator does not prescribe the mechanism to be used for the coordination procedures between the test system and the System Under Test (SUT). Every time when the test script calls for a test coordination action, the translator will generate a call to the routine "implicit_send()" whose function remains unspecified. A simple implementation could issue a message on the test equipment console to request action by the test operator.

Prior to the execution of test cases, the answers to PIXIT questions must be entered into the test system in order to initialize the Test Suite Parameters. The translator generates a routine "tsparameter()" which, when invoked, issues a sequence of I/O requests on the standard input/output device (operator console) for the operator to enter the proper values. It is also expected that this routine will be replaced by a new one that obtains answers to PIXITs from a variety of sources: menu on screen, data on file, etc.

In summary, while the translator provides a prototype of routines **main**, **implicit_sent**, and **tsparameter** for the user (test operator) interface functions, it is up to the test system implementor to design a more user friendly interface mechanism for the operation of the test system.

4.2 Protocol Dependent Support

This support includes facilities for formatting the PDUs from the abstract structure to the real protocol PDU format for transmission to the IUT through the underlying layers, and for comparing PDUs received from the IUT against the abstract structures as defined in PDU constraints. The following is a set of routines (primitives) providing the required facilities. For each PDU **X** defined in the TTCN test script, two primitives are needed:

Send_PDU_X(pointer to pdu constraints) -- This primitive formats a PDU using the input parameter provided and forwards it to the next protocol layer for transmission. An image of the PDU shall also be recorded in the standard trace file.

Is_PDU_X(pointer to pdu constraints, pointer to buffer) -- This primitive analyzes the contents of the buffer and compares it with the values of the abstract PDU structure provided in the input argument; it returns a Boolean value TRUE if every PDU field satisfies the constraints; it returns a FALSE otherwise. The contents of the received buffer must be copied into the corresponding fields of the PDU constraints after the comparison. Before calling this primitive, **Receive_PDU** is presumed to have been invoked to retrieve a message in the input queue and obtain the address of the buffer containing the message retrieved.

A brief discussion on how to write these two set of primitives is given in Section 7.13.5; a sample is also provided in Appendix A.

A similar set of primitives, **Send_ASP_X** and **Is_ASP_X** is also needed if the ATS contains ASP declarations.

4.3 Test Machine Software/Hardware Dependent Support

This support includes all of the facilities required to support the underlying layers of the protocol being tested, as shown in the box labeled (N-1) service supports in Figure 3. If a protocol to be tested is located at layer N, then the supports include the activation and deactivation of software/hardware for layers (N-1) to 1, the movement of N-PDUs through these layers, and the management of First-In-First-Out (FIFO) queues at the N and (N-1) layer boundary. Additional support must be provided for the recording of the execution log, and the management of TTCN timers. The following is a list of primitives referenced by the generated C statements.

Receive_PDU(Pointer to timer-name) -- This primitive removes a message from the top of the FIFO input queue from the IUT (or the N-1 interface), and returns the pointer to the buffer of that message. It also saves the actual length of the message (in number of bytes) in the global variable **_bufferlen**. The input parameter is a pointer to the character string representing the name of a timer. If the queue is empty and if the timer-name is not null, it shall wait for a message to arrive until the specified timer expires. In either case, if no message is available, it returns a null pointer. The buffer space could be reused or released on the next call to **Receive_PDU**.

It is recommended that an image of the PDU received be time stamped and added to the standard trace file, and that if a time out had occurred (i.e. no PDU was received), an empty record be written.

Send_PDU (pointer to buffer) -- This primitive performs the function opposite to the **Receive_PDU** routine, that is, places the message in the output (to the IUT) FIFO queue. This primitive is to be used in the **Send_PDU_X** routines and is not explicitly referenced by the generated C statements; therefore, the exact syntax and semantics rules are not specified here.

Start_timer(timer-name, timer-value) -- This primitive starts the timer specified by time-name with the clock duration set to timer-value in units of milliseconds. Timer-name contains a character string representing the name of the timer.

Cancel_timer(timer-name) -- This primitive cancels the specified timer. If the timer-name is a null string, or if the pointer is 0, all active timers are canceled.

Read_timer(timer-name) -- This primitive returns the time elasped since the specified timer was started. If the timer is not active, a value of 0 shall be returned.

Timeout(timer-name) -- This primitive returns a Boolean value TRUE if the timer specified has expired or had never been started, and a FALSE otherwise. If the input is 0 or points to a null string, then it returns TRUE if any of the active timers has expired.

5. The Test Case Driver

As stated in Section 4.1, at the end of compilation, the TTCN translator generates a main program in C that invokes all of the test cases in the test suite sequentially. The following shows the skeleton of a sample main program. It first calls **tsparameter** to initialize Test Suite Parameters (see Section 4.1 and 7.6), **initialize** to initialize items required by the test system environment support, and **test_suite_init** to initialize items required by the ATS. Examples of test system initialization include activation of communications channels and software for lower layer protocols, and exchange of messages to bring the IUT to a certain pre-test condition. All of these activities are test machine or test suite dependent, and, therefore, must be provided by the system implementor.

The main part of the program consists of a sequence of case statements, one case per test case.

The routine **get_first_tc** selects the first test case to be executed. All subsequent test cases are executed sequentially. For each test case, in addition to the routine for the test case itself, the routines **Output_Trace, Flush_queue,** and **Print_verdict** are also used. The functions of these routines are described below, and prototypes for each routine are also included in the file sysdef.c

```
main()
{       int tcnumber;
        char *msg1="\n\nStarting test case";
        char *msg2 = "End test case";
        tsparameter();
        initialize();
        test_suite_init();
        switch(tcnumber = get_first_tc()) {
        case 0:
           Output_Trace(msg1," PL1_101 ");
           Flush_queue();
           PL1_101();
           Print_verdict(msg2," PL1_101 ");
        case 1:
        ...
        }
}

_tcname _tcnames[] = {
        "PL1_101", 0,
        "PL1_102", 1,
        ...
 };
 int _ntestcases = 450;
```

Flush_queue() -- Removes all messages waiting in the FIFO input/output queues and clears the standard input buffer.

Output_Trace(string,string) -- Sends strings of characters to the standard trace file.

Print_verdict(string,string) - Prints the verdict in **Result** and strings given in the input parameter to the standard trace file.

Finally, the name and number of test cases are saved in the following special data structure at the end of main program:

_tcnames -- A structured array whose entries contain the name of a test case and the test case number in the order of its appearance in the ATS.

_ntestcases -- Contains the total number of test cases in the test suite.

6. The TTCN Language

Version 3 of the NIST TTCN Translator supports the DIS (Teddington) Version of the TTCN Language. It accepts all major features of the TTCN with the following exceptions:

(1) ASN.1 notation is not supported.

(2) The ATS must use only one Point of Control and Observation (PCO).

(3) Constraints references can not be nested, that is, the actual parameters in a constraint reference cannot call another constraint.

(4) The encode and decode functions are not supported.

(5) The test suite constants cannot define Bitstring, Octetstring, or Hexstring constants longer than 32 bits.

(6) Data object references using the array or bit element notation are not supported.

(7) The constraint value for PDU fields cannot contain a range or list of values.

7. Translation of TTCN into C Statements

A detailed description on the mapping of TTCN into the C language is presented in this section. In the discussions below, it is assumed that the standard default options for the Translator are used; more specifically, the outputs are to be sent to files driver.c, suite.h, and suite.c.

7.1 Mapping of Data Structures

The following table shows the mapping of TTCN predefined data types onto the corresponding C data structures.

TTCN type	C Data Type
Integer	long int
Octetstring	a. four octets or less: long int
	b. more than four octets: unsigned array of char
Hexstring	a. eight digits or less: long int
	b. more than eight digits: unsigned array of char
Bitstring	a. 32 bits or less: long int
	b. more than 32 bits: array of unsigned char
Characterstring	array of char
Boolean	int

For the Octet-, Hex-, and Bit-string types, if the length is less than or equal to 32 bits, they are mapped as long integers, otherwise they are mapped into unsigned character arrays. These types are considered numeric in nature, therefore, their contents are right justified, and padded (with zero's) or truncated on the left. Even though TTCN rules do not allow conversion between the above string types and integer, all contstants of these types are considered compatible and interchangeable. For example, the constants 15, 'F'H, '0F'O, and '1111'B are all equivalent.

Character strings are of variable length just like C character strings. End of strings is signaled by a '\0' character in the text, and therefore, the size of the C char array is one (1) greater the length of the TTCN string. Their contents are left justified, and truncated on the right, if necessary.

There are two ways to specify the length of strings in TTCN: fixed length, such as [10], and variable length, such as [1 .. 10]. The numbers inside the bracket [ ] could be a constant, a test suite parameter, a variable, or another PDU field in the same PDU. The translator needs to know the maximum space required, so that an array of sufficient length can be created; therefore, if the length specified is not a constant, a derived maximum length is used by the translator. The derived maximum length is the size of the largest number which could be stored in that length variable. For example, if the length is [1 .. DATASIZE] and if DATASIZE is Octetstring[1], then the maximum length is 256.

According to the DIS Version of the TTCN, a length can only be specified in the PDU fields and user type declarations. However, the TTCN Translator requires that the length be specified in other declarations as well, including test suite parameters, test suite constants, test suite variables, and test case variables.

If the length of a variable or PDU field is not specified, the default length is 32 bits for numeric string types and 80 characters for character strings.

7.2 Translator Generated Variables

The following is a list of variables generated by the translator. All but the variable **_lastlevel** are defined in the driver.c file.

_level -- This is a global variable used to keep track of the indentation level. At the beginning of each test case, **_level** is set to one (1), and is incremented by one each time execution moves down one level.

Result -- This is a global variable, equivalent to the TTCN variable **R**, which holds the preliminary verdict during the execution of the behavior tree and the final result at the termination of a test case. It is initialized to "NONE" at the beginning of each test case.

_buffer -- This is a global variable used to hold the pointer to the last message received from the IUT.

_bufferlen -- This is a global variable containing the length of text (in number of bytes) in the buffer pointed to by **_buffer.**

_timername -- This is a global variable containing the timer name of the first Timeout statement in an indentation level.

_lastlevel -- This is a local variable declared in each behavior tree. It is used to save the current indentation level before entering an attached tree.

7.3 *Test Suite Overview*

No C statements are generated by the C translator for this section.

7.4 *User Type Declarations*

The base type of the new user defined type is recorded, but no C statements are generated. All subsequent uses of the new types are converted to their base type and mapped to the corresponding C structure as described in section 7.1. Two user defined types are considered compatible if their base types are compatible.

7.5 *User Defined Operators*

An external function declaration is generated in the suite.h file for each user operator declared. Since the text defining the operation is presented as FREETEXT in the TTCN.MP, it is ignored by the translator. The system implementor must provide a separate routine just like other primitives supporting the test environment. This must be done even if the TTCN.MP text already defines the body of the operator in a common programming language such as C or PASCAL. All references to the user defined operators are treated as references to external functions. The type of the actual parameters must match the type declared for the corresponding formal parameters.

7.6 *Test Suite Parameters*

The purpose of this section in a TTCN test suite is to associate PICS and PIXIT questions with internal variables in the test suite. The associated values must be entered at the time of system initialization before the beginning of a test session.

The translator generates a C routine, **tsparameter()**, which is to be invoked at the beginning of the test driver program. The purpose of this routine is to prompt the test operator for the values of test suite parameters. The routine contains a set of print and read statements on the standard input/output devices for the parameters declared. The text of the print statements are taken directly from the comments field in the Test Suite Parameter table. In addition, C declarations are generated for names declared therein. It is expected that this routine will be replaced by the system implementor with a more user friendly routine, for example, a menu driven routine.

Example:

Test Suite Parameters		
Name	**Type**	**Comments**
Fixed_TEI Fixed_TEI_No	BOOLEAN BITSTRING[7]	Test fixed TEI only Assigned fixed TEI number

Output in driver.c file:

```
tsparameter()
 { printf("Enter test suite parameters.\n\n");
   printf("Test fixed TEI only. Enter 1 for True and 0 for False.\n");
   scanf("%d", &Fixed_TEI);
   printf("Assigned fixed TEI number.  Enter a hexadecimal string.\n);
   scanf("%x", &Fixed_TEI_No);
 }
```

Output in suite.c file:

```
int Fixed_TEI;
long Fixed_TEI_No;
```

Output in suite.h file:

```
extern int Fixed_TEI;
extern long Fixed_TEI_No;
```

7.7 *Test Suite Constant Declarations*

Test Suite Constants are translated into #define statements. Due to the limitation of the C language, constants for Bit-, Hex-, and Octet-strings are restricted to a maximum length of 32 bits.

7.8 *Test Suite Variable Declarations*

Test Suite Variables are mapped to C data structures as described above. For each variable in the table, a declaration statement is generated in the suite.c file to define the identifier and to initialize its contents, if specified.

Example:

Test Suite Variables			
Name	**Type**	**Value**	**Comments**
TEI_Assigned	BOOLEAN	FALSE	
TEI_Number	BITSTRING [7]	64	TEI # to make call
No_Links	INTEGER		

Output in suite .h file:

```
extern int TEI_Assigned;
extern long TEI_Number; /*TEI # to make call*/
extern no_links;
```

Output in suite .c file:

```
int TEI_Assigned = FALSE;
long TEI_Number = 64;    /* TEI # to make call */
long No_Links;
```

7.9 *Test Case Variable Declarations*

According to the TTCN rules, Test Case Variables (TCV) must be initialized before each test case is executed. Therefore, the translator, in addition to mapping TCVs to the corresponding C data structure just like Test Suite Variables, generates a routine, **testcase_vars_init**(), which is invoked upon entry to every test case. The body of the routine contains assignments or other statements to set the contents of TCVs to the values indicated. If no value is given to a variable, zero (0) is assumed.

Example:

Test Case Variables			
Name	**Type**	**Value**	**Comments**
FLAG	BOOLEAN	TRUE	
I_CNT	INTEGER	0	Counter for no. of I_frames received
MSG_TYPE	OCTETSTRING		

Output in suite.h file:
```
extern int FLAG;
extern long I_CNT; /*Counter for no. of I-frames received*/
extern long MSG_TYPE;
```

Output in suite.c file:

(at variable declarations)
```
int FLAG;
long I_CNT; /*Counter for no. of I-frames received*/
long MSG-TYPE;
```
(at end of declarations)
```
testcase_vars_init ( )
{
        FLAG = TRUE;
        I_CNT = 0;
        MSG_TYPE = 0;
}
```
(at the beginning of every test case)
```
testcase_vars_init ( );
```

7.10 *PCO Declarations*

No C statements are generated for the PCO declarations. The translator keeps track of the name of PCOs and makes sure that event lines in the behavior part are on the same PCO.

7.11 *Timer Declarations*

No C statements are generated. The value of timer duration is used as the default value for the start_timer operation.

7.12 *Abbreviations*

If a test suite contains abbreviations, the TTCN MP file must be processed by a separate pre-processor (see Section 3.2) to fully expand these abbreviations, as the translator does not handle them. The pre-processor provided takes the TTCN MP file as the input and sends output to the standard output file. Therefore, the output could be redirected to a file or piped to the translator. It is recommended that piping be used so that the line numbers in the translated C files refer to the original MP source line number. The expansion applies not only to the event lines in the Behavior Table, but also to every identifier appearing after the abbreviation declaration table.

7.13 *PDU and ASP Declarations*

PDUs and ASPs are translated into C structures whose elements correspond to fields in the PDU or ASP. The mapping of PDU/ASP fields is the same as global variables as described in sections 7.1 and 7.8. A new structured C type is defined for each PDU or ASP, and a variable and a pointer variable of this new type are declared as well. In the following discussion, the word "PDU" applies to "ASP" as well.

7.13.1 PDU Field Tag

Corresponding to each PDU field, a "tag" element will be created in the PDU structure. This is to be used for the PDU constraints, as a field in a constraint. It not only specifies a value or values, it also specifies an operation or option on the value(s). These operations are to be used in formatting PDUs to be transmitted or analyzing PDUs received. They include:

not present ("-") -- the field shall not be present.
don't care ("?") -- contents of the field may be of any value of the type defined for the field.
wildcarded ("*") -- similar to don't care, in addition, the field may not be present.
relational op (">,<,>=,<=,<>,=") --
contents of the field must satisfy the relational operation indicated.
range or list -- the contents of the field contain a range or list of values. This feature is not currently implemented.

Possible values for the tag field, as well as their C parameter names, are shown below. These parameters will be defined in the sysdef.h file with "#define" statements.

Value	Name	Meaning
0	EQUAL_TO	equal ("=")
1	NOT_EQUAL	not equal ("<>")
2	GREATER_THAN	greater than (">")
3	LESS_THAN	less than ("<")
4	GREATER_EQ	greater than or equal (">=")
5	LESS_EQ	less than or equal ("<=")
6	DONT_CARE	don't care ("?")
7	WILD_CARD	wildcarded ("*")
8	OMITTED	not present ("-")
9	RANGE	the value field contains a range of values
10	LIST	the value field contains a list of values

In addition to the C structure described above, two external function prototype declarations for each PDU declaration will also be generated:

```
extern Send_PDU_name(PDU_type *);  and
extern int Is_PDU_name(PDU_type *, char *);
```

where name is the name of the PDU declared in TTCN, and PDU_type is the name of data type generated by the translator.

7.13.2 PDU Field Length

As described in section 7.1, the length of a PDU field could be of variable length determined by the contents of the variables or other PDU field at the instance when the PDU is used. Furthermore, the PDU constraint value can also specify a range of permissible sizes. Due to these requirements, additional structure is needed in the translated C source code to keep track of the actual PDU field length. Very similar to the tag field described in the previous section, a length field is created for each PDU field. The length field contains two subfields carrying the lower and upper bounds. When the length is of fixed size, the two bounds are equal and are set to the actual length. The length is in the unit of the type declared for that field: bytes for Octetstrings, hex-digits for Hexstrings, and bits for Bitstrings.

Since not all test suites require this feature, the translator will generate the length field only if the -l option is specified (see section 3.2). This option applies to every field in all PDUs within a test suite.

7.13.3 PDU Field Group

PDU Field Group tables allow partitioning of a PDU definition into groups, each of which is represented in a separate group table. The format of a PDU Field Group table is identical to the PDU type definition, and, therefore, the sequence of C statements generated are the same in both cases. Since a Group table may again refer to other Group tables, a PDU definition may be mapped into a nested C structure.

7.13.4 Examples of PDU Type and PDU Field Group Declarations

PDU Type Declaration		
PDU Name: DATA	**PCO Type:** NSAP	**Comments:**
PDU Field Information		
Field Name	**Type**	**Comments**
HEADER	GROUP	Packet header
P_S	INTEGER	Send sequence no.
P_R	INTEGER	Receive sequence no.
M	BITSTRING [1]	More bit
INFO	OCTETSTRING [1 .. 4096]	Max. 4096 bytes

PDU Field Group Type Declaration		
Field Group Name: HEADER	**Comments:** Packet Header	
PDU Field Information		
Field Name	**Type**	**Comments**
QBIT	BITSTRING [1]	
DBIT	BITSTRING [1]	
GFI	BITSTRING [2]	
LCI	HEXSTRING [3]	Logical channel id

Output in suite.h file:

```
typedef struct { int from; int to } _LENGTH;

typedef struct {
          long QBIT;
          long DBIT;
          long GFI;
          long LCI; /* logical channel id */
          char QBIT_tag, DBIT_tag, GFI_tag, LCI_tag;
          _LENGTH QBIT_len, DBIT_len, GFI_len, LCI_len;
}         group_HEADER;
```

```
typedef struct {
          group_HEADER HEADER;
          long P_S; /* Send sequence no. */
          long P_R; /* Receive sequence no. */
          long M; /* More bit */
          BYTE INFO [4096]; /* Max. 4096 bytes */
          char P_S_tag, P_R_tag, M_tag, INFO_tag;
          _LENGTH P_S_len; P_R_len, M_Len, INFO_len;
}         pdu_DATA;
extern Send_PDU_DATA ();
extern Is_PDU_DATA ();
extern pdu_DATA DATA, *pDATA;
```

Output in suite.c file:

```
pdu_DATA DATA, *pDATA=&DATA;
```

Note that lines in *italic* are generated only if the -l option is specified.

7.13.5 How to Write Protocol Support Primitives

As mentioned in Section 4.2, a system implementor must provide two primitives for each PDU declared in a test suite: Send_PDU_* and Is_PDU_*. In these routines, the tag of each PDU field must be examined to determine the type of comparison or operations to be used for the field. For the Send_PDU_* routine, the tag field can only be one of the following: EQUAL_TO, DONT_CARE, and OMITTED. In the first two cases, the contents of the PDU constraint are copied into the buffer, and for the third case, the field is omitted. All mandatory fields defined in the protocol, but not declared in the TTCN PDU, must be filled with their mandatory values. For example, a test suite may not include in its PDU field declaration the frame control field indicating the type of frame, because its content is implied by the name of the PDU.

For the Is_PDU_* routines, all types of tag values for each PDU field are possible, and, therefore, a case statement (switch) is needed to consider all possibilities. In addition to satisfying the condition indicated in the tag, especially for the cases of DONT_CARE and WILD_CARD, the field value must conform to the type of the field. For example, if the type of a field is a user defined type consisting of three possible values, say 1, 3, and 5, then the value in the received buffer must contain one of these values, even if the tag for the field is DONT_CARE. Those mandatory fields which are not declared in the PDU type declaration must still be checked with their mandatory values. Furthermore, the actual values received must be copied into the C structure for the constraints.

Appendix B contains a sample coding of these routines.

7.14 Constraints Declaration

7.14.1 PDU and PDU Field Group Constraints

For each constraint name associated with a PDU, a C function with the same name is generated.

This function assigns values to each of the PDU fields and their associated tags and lengths (see Section 7.13). If the field refers to a group name, then a call to the group constraint name referenced in the value column is generated. The formal parameters of the constraints are converted to a structured type called CONSTARG, which is described in Section 7.1.7.2.

Example:

PDU Constraints Declaration	
PDU Name: DATA	**Constraint Name:** UD(PS_NO,PR_NO:INTEGER, LCI:OCTETSTRING[3])
Field Name	**Value**
HEADER	HDR1(LCI)
P_S	PS_NO
P_R	PR_NO
M	0
INFO	'C1C2C3C4'O[4]

PDU Field Group Constraint Declaration	
Field Group Name: HEADER	**Constraint Name:** HDR1(LCI_NO:OCTETSTRING[3])
Field Name	**Value**
QBIT	0
DBIT	0
GFI	'01'B
LCI	LCI_NO

Output in suite.c file:

```
HDR1(p_HEADER,LCI_NO);
        group_HEADER *p_HEADER;
        CONSTARG *LCI_NO;
{
        p_HEADER->QBIT = 0; p_HEADER->QBIT_tag = EQUAL_TO;
        p_HEADER->QBIT_len.from = 1; p_HEADER->QBIT_len.to = 1;
        p_HEADER->PBIT = 0; p_HEADER->PBIT_tag = EQUAL_TO;
        p_HEADER->PBIT_len.from = 1; p_HEADER->PBIT_len.to = 1;
        p_HEADER->GFI = 0x1; p_HEADER->GFI_tag = EQUAL_TO;
        p_HEADER->GFI_len.from = 2; p_HEADER->GFI_len.to = 2;
        p_HEADER->LCI = LCI_NO->value.v; p_HEADER->LCI_tag = LCI_NO->tag;
        p_HEADER->LCI_len.from = LCI_NO->len.from;
        p_HEADER->LCI_len.to = LCI_NO->len.to;
}
```

```
UD(p_DATA,PS_NO,PR_NO,LCI)
        pdu_DATA *p_DATA;
        CONSTARG *PS_NO;
        CONSTARG *PR_NO;
        CONSTARG *LCI;
{
        HDR1(&(p_DATA->HEADER),LCI);
        p_DATA->P_S = PS_NO->value.v; p_DATA->PS_tag = PS_NO->tag;
        p_DATA->P_S_len.from = PS_NO->len.from;
        p_DATA->P_S_len.to = PS_NO->len.to;
        p_DATA->P_R = PR_NO->value; p_DATA->PR_tag = PR_NO->tag;
        p_DATA->P_R_len.from = PR_NO->len.from;
        p_DATA->P_R_len.to = PR_NO->len.to;
        p_DATA->M = 0; p_DATA->M_tag = EQUAL_TO;
        p_DATA->M_len.from = 1; p_DATA->M_len.to = 1;
        bitcpy(&(p_DATA->LCI),32760,cvttobit("C1C2C3C4",32),32);
        p_DATA->LCI_tag = EQUAL_TO;
        p_DATA->INFO_len.from = 4; p_DATA->INFO_len.to = 4;
}
```

The lines in *italic* are generated only when the translator -l option is specified.

7.14.2 Modified Constraints and Base Constraints

A constraint may modify a base constraint by specifying only those fields whose values are different from the base constraint. The constraint is called a modified constraint; the base constraint itself may be a modified constraint. Any fields not specified will default to those in the base constraint. Given a modified constraint, C0.C1.C2, the translator will generate a routine, C2, that references its base constraint, C1. The constraints C0, C1, and C2 must be defined in the order given, that is, C0 first, C1 next, and C2 last. The skeleton of the routine C2 is shown below. The routine, C1, will have similar structure and will reference C0.

```
C2(parameters)  <-- parameters consists of all parameters for C0, C1, and C2 combined.
{
  C1(parameters); <-- all parameters needed for C0, and C1
  Assignments to fields defined in C2;
}
```

7.15 <u>Behavior Trees in Dynamic Behavior Part</u>

A C function (routine) is generated for each behavior tree in the Dynamic Behavior Part. The routines for test case trees are to be invoked from the test case driver. All others are called by attachment event from test cases or test steps. If, during the execution of the function, a final verdict is assigned, the function will terminate and return a Boolean value of TRUE, indicating that the execution should be terminated. Otherwise, a value of FALSE will be returned. The verdict is stored in the global variable named **Result**.

The names of C functions are the same as the corresponding tree names. This may cause problems on test machines that limit the length of external names.

Example: The following statements are generated for the behavior tree with the following header:

TEST_State(X:INTEGER; Y:OCTETSTRING)

```
int TEST_State(X,Y)
  int X; long Y;
  { int _lastlevel;    /* other declarations may follow */

     ... /* body of tree here */

    return (FALSE);    /* generated only if no verdict was assigned */
  }
```

The size of a formal parameter, if not declared, is determined from two sources: (1) outside of the tree -- the actual parameter in the first reference; (2) inside the tree -- the source of the first assignment to the formal parameter. The first one is used if the tree is referenced before its definition.

For the main tree of a test case, the generated function will contain additional statements to initialize global variables. A typical sequence is shown below:

```
int TestCaseName()
{ int _lastlevel;
  _level = 1;
  strcpy(R,"NONE");
  Result = NONE;
  ... /* body of the main tree */

}
```

7.16 *TTCN Event Lines in Behavior Trees*

A TTCN behavior tree basically consists of a set of branches which are represented as a list of event lines with different indentation levels. Branches on the same level are represented by event lines with the same indentation number. Since the branches represent a set of mutual exclusive alternatives, the corresponding TTCN event lines are translated into the C language if ... else ... structure. The TTCN semantics requires that once an indentation (branch) level is entered, the alternative events in that level must be checked infinitely until one of the events occurs, then the evaluation of the tree either continues at the next level or terminates. The corresponding C structure, therefore, is an infinite while loop with an if statement as the body of the loop:

```
while (TRUE)
{ if ...
  else if ...
        else ...
}
```

The evaluation of a TTCN tree is terminated when an event line assigns a final verdict or when an end branch is reached. A return statement is generated after these event lines. The presence of a return statement at the end of a "while TRUE" loop may cause some C compilers to flag the statement following the loop as unreachable; therefore, the statement "while (_forever)" is used instead. The variable _forever is initialized to TRUE in the declaration.

Example: The event lines of a partial behavior tree and its corresponding C structure are shown side by side below.

```
E1                  entry code for indentation level at E1
                    while (_forever)
                    {    evaluation of fields in PDU constraints for E1
                    if (event E1 and Boolean expressions for E1)
                    { statements for assignments in E1;
        E11           entry codes for indentation level at E11;
        E12           while (_forever)
        ...           { statements for E11, E12, and ...; }
                    }
E2                    /* beginning of line E2 */
                    evaluation of fields in PDU constraints for E2;
                       if (event E2 and Boolean expressions for E2)
                       { assignment in E2;
        E21              entry codes for indent level at E21;
        ...              while (_forever)
                         {  statements for E21 and ...; }
                       }
E3                    /* beginning of line E3 */
        ...         E3 and ...
...
                    }
```

7.16.1 *Indentation Level Entry Codes*

Upon entry into a new indentation level, some housekeeping operations must be done, therefore a sequence of "entry codes" is generated. The statements in the sequence depend on the indentation level.

A. Entry codes for the first indentation level:

1). the statement "_level = 1;" to initialize the level nesting counter;
2). if there is a TIMEOUT event in this indentation level, the statement
    if (_timername == 0) _timername = "timer-name";
   where timer-name is the name appearing in the first TIMEOUT statement;

B. Entry codes for the second indentation level or higher:

1). the statement "_level += 1;" to increment the level nesting counter;
2). if there is a TIMEOUT event in this indentation level, the statement

_timername = "timer-name";
where timer-name is the name appearing in the first TIMEOUT event;

3). if there is a tree attachment (including the Repeat event), but no receive event in this indentation level, the statement "_bufferlen = -1;" to indicate that nothing has been removed from the input FIFO queue yet;
4). if there is no TIMEOUT event appearing in this level, but there is a tree attachment or a receive event, the statements "_timername = (char *) 0;"
5). the first half of the while statement -- "while (_forever) {".

Since the first indentation level of a subtree is considered to be at the same level where the tree is attached, and the initialization for that level has been done before the attachment, therefore, some of the entry codes must be omitted or conditionally executed. The condition on item A.2), above, allows the timername to be set only if there was no TIMEOUT event in the calling tree.

7.16.2 Verdict in an event line

When an event line assigns a verdict, the translator generates a call to one of the two pre-defined functions (or macros), **SetpVerdict**(new-verdict) and **SetfVerdict**(new-verdict), for the preliminary and final verdicts respectively, to set the verdict in the global variable **Result**. If the verdict is a preliminary verdict, SetpVerdict will also store the result in a pre-defined TTCN character string variable **R**; if it is a final verdict, the translator will generate a return(TRUE) statement, which will terminate the execution of the function.

7.17 Send Events

Given a send event line of the form

!P [Boolean expressions] (Assignment clauses) P[C]

where P is the name of a PDU and C is the constraint associated with the PDU P, the translator generates a sequence of statements in the following order:

- statements to prepare the actual arguments for the constraint C;
- call C() to evaluate the constraint C;
- an if statement to evaluate Boolean expressions;
- assignment statements for the Assignment clauses;
- call Send_PDU_P to transmit the PDU.

7.17.1 Evaluation of Boolean Expressions and Assignment Clauses

The translation of Boolean expressions and assignment clauses is straight forward -- an expression is either copied without change or transformed into a valid C expression. References to PDU fields will be transformed into C pointer notation. For example, let XYZ be a PDU name, an identifier of the form XYZ.field is transformed into pXYZ->field where pXYZ is the pointer to the storage for PDU structure XYZ.

Arithmetic operations are not allowed with operands longer than 32 bits which are mapped into C unsigned character arrays. Relational and assignment operations involving arrrays are translated into calls to one of the following predefined functions:

> int **strcmp**(char *S1,char *S2) -- for character compare
> int **bitcmp**(char *S1,int length-of-S1,char *S2,int length-of-S2) -- for numeric string compare
> void **strcpy**(char *S2,char *S1) -- for copying character string S1 into S2
> void **bitcpy**(char *S2,int length-of-S2,char *S1,int length-of-S1) -- for copying numeric string S1 into S2.

These functions are provided in the sysdef.c file.

7.17.2 Evaluation of PDU constraints references

Since PDU constraints are translated into C functions, evaluation of constraints references consists of a C function call to the referenced constraint.

If a constraint contains arguments, then instead of passing the argument itself, the translator will generate a pointer to a special argument structure containing either the value, or a pointer to a list of values, as well as the tag, and optionally the length of values. The special argument structure is declared in suite.h as follows:

```
typedef struct {
   union { long v,  char *ptr} value;
   char tag;
   _LENGTH len;
} CONSTARG;
```

For example, let C1(X,Y) be a constraint for PDU ABC with two arguments, X being a long integer and Y being an unsigned character array of size 10 (Octetstring[10]), the constraint reference will be translated into

```
c_arg1.value.v = X; c_arg1.tag = EQUAL_TO;
c_arg1.len.from = 1; c_arg1.len.to = 1;
c_arg2.value.ptr = Y; c_arg2.tag = EQUAL_TO;
c_arg2.len.from = 10; c_arg2.len.to = 10;
C1(p_ABC,&c_arg1,&c_arg2);
```

where c_arg1 and c_arg2 are declared at the beginning of the suite.c file as:

```
CONSTARG c_arg1, c_arg2;
```

The lines in *italic* are generated only if the translator -l option is specified.

7.17.3 Transmission of PDUs

It is assumed that the test implementor will provide a primitive to transmit PDUs to the IUT. The function call to use the primitive is of the form

```
Send_PDU_name(pointer to PDU constraint);
```

Example: To transmit the constraints C1 for PDU ABC evaluated by the sequence shown in the previous section, the translator would generates:

```
Send_PDU_ABC(pABC);
```

7.17.4 Verdict on the Send Event

The following sequence shows the translation of a send event without a final verdict or with a preliminary verdict:

```
evaluation of constraints
assignment clauses
if (Boolean expression) {
     Send_PDU_name(pointer to PDU constraints);
     SetpVerdict(verdict); /* generated only if there is a preliminary verdict */
             entry codes for the next level of behavior tree
             statements for the next level
                     return(TRUE) or return(FALSE)
}
next alternative at the same level
```

If an event line contains a final verdict, the following sequence is generated instead:

```
evaluation of constraints
assignment clauses
if (Boolean expression) {
     Send_PDU_name(pointer to PDU constraints);
     SetfVerdict(verdict);   return(TRUE);
}
next alternative at the same level
```

7.18 Receive Events

The translation of a receive event line is similar to the send event line. The following sequence of statements is generated:

- an if statement referencing Is_PDU-name(pointer to constraints, pointer to buffer) to analyze the incoming PDU;
- additional expressions in the if statement for boolean expressions in the event line;
- assignment statements in the assignment clause of the event line.

In addition, for the first receive event line at the indentation level two or higher, a call to

Receive_PDU to read the input from IUT is also generated:

```
    _buffer =  Receive_PDU(_timername);
```

For the first indentation level of a subtree, a conditional call to Receive_PDU is generated instead, since an input record may have already been read at the level where the subtree is attached.

```
    if (_bufferlen < 0) _buffer = Receive_PDU(_timername);
```

Note, that a negative value in _bufferlen indicates that the Receive_PDU routine has not been executed for the current indentation level. The following is a complete sequence of statements generated for the receive event.

```
evaluation of constraints
if ((Is_PDU_name(pointer to PDU constraint, _buffer)) &&
    (Boolean expressions) )
{
     assignment clauses
     SetpVerdict(verdict) or SetfVerdict(verdict);
             /* generated only if there is a verdict */
     return(TRUE); or
             entry codes for the next level of behavior tree
             statements for the next level
}
next alternative at the same level
```

7.19 *Implicit Send*

The implicit send event line is used to specify the coordination procedures needed to trigger actions from the upper layer of IUT. It is of the form

```
<IUT!PDU> comments
```

It will be translated into a call to a special routine **Implicit_send**, as defined below.

```
Implicit_send(message)
  char * message;
{  printf("Coordination action, force IUT to send %s", message);
}
```

The input parameter points to a string containing the PDU name and the comments in the implicit send event line.

This routine could be modified for other test coordination methods. For example, instead of printing on the console, the routine could send the message through a communication line connecting to a test coordinating program on the SUT, which in turn would force the desired action on the IUT.

7.20 The OTHERWISE Event

Any input message from the IUT will match the OTHERWISE event, therefore, all that is needed is to test if the buffer pointer returned by the Receive_PDU primitive is null, as shown below:

```
if ( _buffer ) { ... /* actions for OTHERWISE */}
```

7.21 The ATTACH Event

Tree attachments are translated into function calls. The attached tree (function) is expected to return a Boolean value to indicate whether the execution of the test step should continue or not. If the function returns a value FALSE, then the execution of the tree will continue either at the same indentation level where the tree is attached or at the next level, depending on how far the execution of the function has traversed the attached tree. If the function returns from the first indentation level in the attached tree, then all events at this level did not match and, therefore, the execution should continue on the next alternative following the tree attachment; if the function returns from one of the leaves of the attached tree, then at least one event matched and, therefore, the execution should continue at the next level of that tree attachment.

If the function returns a value true, it indicates that a final verdict has been assigned, and that the execution of the tree should be terminated.

Example. The following TTCN sequence

```
+Subtree(m,n)
    x
    ...
y
...
```

is translated into:

```
_lastlevel = _level;
if (Subtree(m, n)) return (TRUE);
else if (_level > _lastlevel)
  {  /* test script at the next indentation level */
    _level = _lastlevel + 1;
    other entry code for this level;
    codes for event x
    ... /* there will be return statements to exit */
  }
_level = _lastlevel;
/* test script at the same indentation level */
codes for event y
...
```

7.22 Timer Management

7.22.1 Timer Operations -- Start, Cancel, Readtimer, and Timeout

Since the timer management depends on the hardware and software implementation of clocks on the target test machine, timer operations are translated into function calls to the appropriate primitives to be provided by the test system implementor. The input to these primitives is the name of the timer, such as "T200". In addition, the duration of the timer, scaled to units of milliseconds, is also passed to the Start_timer primitive. If the TTCN start timer event line does not specify the timer duration, the default value in the timer declaration table is used.

Examples: The translation of the following timer operations are shown on the right of the TTCN statements. It is assume that the default duration of the timere T1 is 2 seconds and that the timer unit for T200 is in seconds.

START T200 (T200value+delta)	Start_timer("T200",(T200value+delta) * 1000);
START T1	Start_timer("T1", 2 * 1000);
CANCEL T1	Cancel_Timer("T1");
CANCEL	Cancel_Time("");
TIMEOUT T1	if (Timeout("T1")) { ... }

7.22.2 Timers in the Receive_PDU primitives

The TTCN snapshot semantics requires that during the execution of a TTCN test script, events within an indentation level be polled continuously until one of the event occurs, e.g., timer expiration or arrival of a message from IUT. For this reason the translated C source code consists of an infinitive while loop for each indentation level. To avoid this repeated polling, the name of the timer used in the indentation level is also passed to the Receive_PDU primitive, which will not return control to the caller until either a PDU is received or the timer has expired. Thus, only one pass through the loop is necessary.

If the indentation level does not have a TIMEOUT test, or contains more than one TIMEOUT tests, then the timer name in the call to Receive_PDU is set to null. In that case, it returns immediately with or without a PDU.

7.23 Labels and the Goto Statement

Labels and Gotos are translated into their corresponding C counterparts. However, labels on entries that are not the first alternatives at a given indentation level are ignored, as jumps to these labels are invalid. A label is always placed before the entry codes (see section 8.2.2.) for the indentation level where the label is defined.

Example: A simple case of goto.

```
                                          L1: _level += 1;
                                           /* other entry codes for level e1 */
                                          while (_forever)
      ? e1               L1               { if (e1)
               ! e11     L11                 { L11: entry codes and body of e11 }
      ! e2                                   if (e2)
                                               { L21: entry codes for level e21
                                                   while (_forever)
               ? e21     L21                       {  if (e21)
                                                            { ... }
               -> L1                                  goto L1;
                                                   }
                                               }
                                          }
```

Even though the TTCN syntax allows jumps from a locally defined test step into the main tree, the translator will not accept this type of goto.

7.24 ***The REPEAT Statement***

The Repeat event line is translated into the C do while structure.

Example: REPEAT Subtree(m,n) UNTIL [Boolean-expression]

C statement

```
do
  if (Subtree(m,n)) return (TRUE);
while (!Boolean-expression)
```

7.25 ***The Default Reference***

If the default tree is referenced, a tree attachment referencing the default tree is generated at the end of each indentation except when the default attachment can not be reached because the last event line was a Goto, an unconditional send event, or a pseudo event with only assignment clauses.

8. Sample TTCN Test Script and Compiled C Statements

Appendix A contains an example showing the translation of a TTCN test suite into C source statements. The TTCN scripts are shown first in tabular format followed by scripts in machine processable form (TTCN.MP) and the compiled C source statements.

9. References

[1] *Information Processing Systems*, OSI Conformance Testing Methodology and Framework - Part 1-5, 1989. The TTCN language is defined in Part 3: The Tree and Tabular Combined Notation, DIS 9646-3, December 1, 1989. (ISO/IEC JTC 1/SC 21)

[2] R.L. Probert, H. Ural, and M.W.A. Hornbeek, "A comprehensive software environment for developing standardized conformance test suites," *Computer Networks*, vol. 18, no. 1

[3] *The ITEX Test Suite Development, An Introductory Overview*, Swedish Telecom, Sweden, 1988.

[4] S. Johnson, "YACC: Yet Another Compiler-Compiler," *UNIX Programmer's Manual*, April, 1986.

[5] M. Lesk, E. Schmidt, "LEX - A Lexical Analyzer Gernerator," *UNIX Programmer's Manual*, April, 1986.

[6] A. Schreiner, H. Friedman, *Introduction to Compiler Construction with UNIX*, Parentice-Hall, Inc., 1985.

[7] C. Donnelly, R. Stallman, *BISON, The YACC-compatible Parser Generator*, Free Software Foundation, Inc. October, 1988.

Appendix A. Sample Primitive Routines for LAPD SABME Frame Send_PDU_SABME and Is_PDU_SABME

Definition of PDU SABME in suite.h File

```
typedef struct
{ /* Set Asynchronous Balanced Mode Extended  */
        long SAPI ;
        long C_R ; /* Cmnd / Resp  */
        long EA_0 ;
        long TEI ; /* TEI number  */
        long EA_1 ;
        long P ; /* poll bit  */
        char SAPI_tag, C_R_tag, EA_0_tag, TEI_tag, EA_1_tag, P_tag;
} pdu_SABME;
extern Send_PDU_SABME();
extern int Is_PDU_SABME();
```

The Send_PDU_SABME and Is_PDU_SABME Routines

```
#include "suite.h"
#include "sysdef.h"

/*
 *  Send_PDU_SABME -- format a SABME frame and forward it for transmission
 */


Send_PDU_SABME(frame)
   struct pdu_SABME *frame;
{
   unsigned char buffer[3];            /* buffer for the SABME frame */

        /* format a SABME buffer for transmission */
        /* Since all these fields are mandatory, no check for tags is necessary */
   buffer[0] = (frame->SAPI << 2) | (frame->C_R << 1); /* first byte */
   buffer[1] = (frame->TEI << 1) | 1;  /* 2nd byte, TEI and EA_1 fields */
   buffer[2] = (frame->P << 4) | 0x6f; /* P and SABME frame control */

        /* now use the layer 1 support routine to send the frame */
  Send_PDU(buffer);                    /* this routine is system dependent */

        /* generate test records */
  trace_send(buffer);                  /* send a trace entry in log file */
        /* we may also want to send a formatted output to
               test console */
}


/*
 *  Is_PDU_SABME -- test if the input in buffer matches the constraints in frame,
 *    return TRUE if the contents match.
 *    Copy contents of buffer to the structure for the SABME frame if necessary.
 */


int Is_PDU_SABME(frame,buffer)
   struct pdu_SABME *frame;            /* buffer must match this frame */
```

```
    unsigned char *buffer;              /* input buffer, assume that layer 1
                          support has already strip the flags and CRC bytes */
{
    extern int _bufferlen;
    if (_bufferlen != 3) return (FALSE);  /* message too short */
    if (( *(buffer+2) & 0xef) != 0x6f)   /* check for SABME frame control */
                 return FALSE;           /* not a SABME frame */

        /* now check the SAPI field based on the relational operation
         indicated in the tag field */
    switch(frame->SAPI_tag) {
          case DONT_CARE:
          case WILD_CARD:
                 break;
          case EQUAL_TO:
                 if (frame->SAPI != (*buffer >> 2)) return FALSE;
                 break;
          case NOT_EQUAL:
                 if (frame->SAPI == (*buffer >> 2)) return FALSE;
                 break;
          case GREATER_THAN:
                 if (frame->SAPI <= (*buffer >> 2)) return FALSE;
                 break;
          case LESS_THAN:
                 if (frame->SAPI >= (*buffer >> 2)) return FALSE;
                 break;
          case GREATER_EQ:
                 if (frame->SAPI < (*buffer >> 2)) return FALSE;
                 break;
          case LESS_EQ:
                 if (frame->SAPI > (*buffer >> 2)) return FALSE;
                 break;
          default: return FALSE;
                 break;
    }
    frame->SAPI = *buffer >> 2;         /* copy the SAPI field into frame */

           /* check command response field -- 1 bit */
    if (frame->C_R_tag == DONT_CARE)
         frame->C_R = (*buffer >> 1) & 1; /* copy the C_R field */
    else {
         if (frame->C_R != ((*buffer >> 1) & 1)) return FALSE;
    }

         /* check the EA0 flag, must be 0 */
    if (*buffer & 1) return FALSE;

         /* next check the TEI number field */
    switch(frame->TEI_tag) {
          case DONT_CARE:
          case WILD_CARD:
                 break;
          case EQUAL_TO:
                 if (frame->TEI != (*(buffer+1) >> 1)) return FALSE;
                 break;
          case NOT_EQUAL:
                 if (frame->TEI == (*(buffer+1) >> 1)) return FALSE;
                 break;
          case GREATER_THAN:
                 if (frame->TEI <= (*(buffer+1) >> 1)) return FALSE;
                 break;
          case LESS_THAN:
                 if (frame->TEI >= (*(buffer+1) >> 1)) return FALSE;
                 break;
          case GREATER_EQ:
                if (frame->TEI < (*(buffer+1) >> 1)) return FALSE;
                break;
        case LESS_EQ:
                if (frame->TEI > (*(buffer+1) >> 1)) return FALSE;
                break;
        default: return FALSE;
                break;
    }
    frame->TEI = *(buffer+1) >> 1;              /* copy the TEI field into frame */

        /* check the EA1 flag, must be 1 */
    if ((*buffer & 1) != 1) return FALSE;

        /* check the Poll bit */
    if (frame->Poll_tag == DONT_CARE)
        frame->Poll = (*(buffer+2) >> 4) & 1; /* copy the Poll field */
    else {
        if (frame->Poll != ((*(buffer+2) >> 4) & 1)) return FALSE;
    }
    return TRUE;  /* all fields matched and copied */
      /* note: before returning, you may want to enter a trace entry */
}
```

Appendix B. UNIX ttcnc Command Manual

NAME

ttcnc - invoke the ttcn compiler

SYNOPSIS

ttcnc <u>mp-filename</u> [-m <u>filename</u>] [-h <u>filename</u>]
[(-s|-g|-t) <u>filename-prefix</u>] [-l]

DESCRIPTIONS

ttcnc is the compiler that translates the ttcn test scripts in the MP form into the C language. All command arguments are optional. If the input filename is not specified, the standard input file is used. The default output files are described in the associated options. All scratch files are deleted at the end of compilation unless terminated prematurely.

OPTIONS

All options described below, except -l, are specifications for output files; all filenames specified will be automatically appended with either a .c or .h suffix as appropriate.

-m : Specify the name of the file for the main routine from which the test cases are invoked. The default file is "driver.c"

-h : Specify the name of the file for the data structure header file. If this option is omitted, the filename specified in the -s option is used; if both options are omitted, the default name is "suite.h"

The next three options, -s, -g, -t, specify the grouping of the test scripts, and are mutually exclusive.

-s : All output, except the main driver and the header file, are to be stored in one file.

-g : The test cases are to be grouped based on the test suite structure, and stored in separate files. The filenames are created by concatenating the prefix specified, and the group name in the test suite. (See next selection for more details.)

In addition, the following files are created (the asterisks represent the prefix):

*decl.c - declarations for variables
*cons.c - routines for the TTCN constraints part
*lib.c - routines for the test step library

-t : Each test case is stored in a separate file. The filename is the concatenation of the prefix and the test case identifier. Just like the -g option, files *decl.c, *cons.c, and *lib.c will also be generated.

If none of the -s, -g, -t options are specified, -s is implied and all output will go to "suite.c".

-l : For each PDU field, provide space for the length of the field. (See section 3.3 for explanation.)

NOTES

If the test suite uses abbreviations, a preprocessor must be used to expand the abbreviated strings. In this case, the shell command is as follows:

```
prep mp-source ¦ ttcnc options
```

EXAMPLES

```
ttcnc lapd.mp
```

Compile the TTCN test suite in lapd.mp and produce files suite.h, suite.c and driver.c.

```
ttcnc lapd.mp -m lapdtest -g lapd
```

Compile the TTCN test suite in lapd.mp and produce the following files: suite.h lapdtest.c, lapdedcl.c, lapdcons.c, lapdlib.c, and one lapd.c per group.*

```
prep x25.mp | ttcnc
```

Expand abbreviations in test suite X25.mp and compile it to produce output files suite.h, suite.c, and driver.c.

NIST-114A
(REV. 3-89)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4655

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
August 31, 1991

4. TITLE AND SUBTITLE
Users' Manual for the NIST TTCN Translator Version 3.0

5. AUTHOR(S)
David H. Su

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)
U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

10. SUPPLEMENTARY NOTES

☐ DOCUMENT DESCRIBES A COMPUTER PROGRAM; SF-185, FIPS SOFTWARE SUMMARY, IS ATTACHED.

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

Standard conformance testing is the first step in ensuring the interoperability of communications products. There has been increased interest in the development of Abstract Conformance Test Suites (ATS) in the standards community. A standard test script language called Tree and Tabular Combined Notation (TTCN) has increasingly been used to specify test suites. NIST has developed a TTCN to C language translator to help the industry in speeding up implementation of ATSs into executable test suites (ETS). This manual describes processes involved in implementing an ETS, how the translator is to be used, and the design of the translator itself.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

abstract test suite; conformance testing; data communications; executable test suite; protocol standard; translator; TTCN

13. AVAILABILITY

[X] UNLIMITED
[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
40

15. PRICE
A03

ELECTRONIC FORM